Resource Management in Message Passing Environments*

Ivan Zoraja[1], Ursula Seitz[2], Arndt Bode[2], Petar Slapničar[1]

[1]FESB, Department of Electronics and Computer Science University of Split, Croatia
[2]LRR-TUM, Institut für Informatik, Technische Universität München, Germany

This paper discusses the need for resource management support for parallel applications running on workstation clusters and communicating by message passing among tasks. Many resource management systems are only able to start a message passing runtime environment and parallel applications, but dynamic reconfiguration fails because of the missing cooperation between the resource manager and the runtime environment. In order to utilize computational resources in message passing environments efficiently, to control execution of parallel applications by rescheduling tasks at runtime, and to minimize their execution time, a resource management system has been developed and preliminary tests results have been carried out. Most of our efforts in this regard have been to design an efficient approach to load measurement and process scheduling and implement the resource management system in a manner such that it can easily be adapted to any message passing framework. Although our first version is based on the PVM system, we also intend to implement an MPI – based resource management system.

Keywords: resource management, load balancing, process migration, message passing, PVM, MPI, workstation clusters.

1. Introduction

Although hardware power incessantly increases, there are always applications which still require a larger amount of computing capacity. Parallelization is a way to shorten the computation time of long running and resource intense applications by dividing the underlying data region or decomposing the application's functions and computing each of the resulting smaller modules by a separate process on different processors which, without having shared memory segments, perform communication by sending messages. Especially appropriate for parallelization are scientific computing applications because they often exceed the resource availability of a single host and the underlying computational grid can easily be partitioned into smaller modules.

Owing to its very high aggregate performance, massively parallel systems (MPPs) have been built to run high performance parallel applications, but, their widespread use has been prohibited by their high price and poor cost-performance ratio. Recently, networks of workstations (NOWs) have been used as a unified concurrent computing resource and have evolved into a very effective and tenable environments for both high performance scientific computation and commercial and business general based data processing. These computing environments are typically based upon hardware consisting of a collection of heterogeneous workstations interconnected by a high speed local area network and are able to achieve a supercomputer's performance at significantly less cost. The most common programming paradigms are those that provide a process or thread-based message exchange among tasks that reside in different address spaces.

The execution of a parallel application is inherently more complex than of a sequential one, because more than one host is involved in the computation and more than one thread of con-

* This work has been funded by the German Federal Department of Education, Science, Research and Technology (**BMBF**) within the research project **SEMPA** and by the Croatian Ministry of Science via the project **FeedBack**.

trol must be started and terminated. A message-passing environment primarily supports the message exchange between the processes, but beyond that, it also performs some resource management functionalities, e.g. the configuration of the runtime environment that controls the hosts and the running processes of a parallel application. However, the resource management functionality in current message-passing environment is rather poor and requires great support by the user.

In a NOW, a resource management system controls host pool and applications and tries to assign idle resources to an application waiting to be computed. Usually, a resource management system works in combination with a batch queuing component, i.e. the user specifies the resources needed for the execution, e.g. machine architecture, and submits the application as a batch job to the resource management system where the job is queued until resources are available. The resource manager also decides on rescheduling or migration in case the system is overloaded or an interactive user has logged in. There are various resource management systems available, e.g. CODINE [6] Condor [12], LSF [8] but most of these systems have been designed for sequential (uniprocessor) applications and lack some functions that are important for parallel applications [10].

In essence, there is no interaction between the resource management system and the message passing runtime system, which prevents both the resource management system of having full control over parallel applications and the parallel application of using resource management functionalities. The most suitable and general way to get resource management and message passing systems connected is to define a layer between them with interfaces that could be adapted to various resource management systems and message passing environments. In this paper, we address the problems resulting from the design and implementation of the SEMPA Resource Manager which connects the CODINE system with the PVM system [22, 1] and includes the CoCheck checkpointing facilities and process migration mechanisms [20, 21].

The remaining article is structured as follows. Section 2 compares message passing systems and libraries such as PVM and MPI [3, 4, 7] and their suitability for a resource management implementation. Since our first implementation is based on PVM, some background material on PVM has been included. The aims and techniques of resource management systems in a NOW are given in section 3. The SEMPA Resource Manager , an example for a resource management system that closely cooperates with the PVM message passing environment, is explained in section 4. Section 5 shows some performance measurements with the SEMPA Resource Manager . Discussion and an outlook on future work are presented in section 6; section 7 closes the article with a brief summary.

2. Message Passing Paradigms

According to the way processors communicate with one another, hardware architectures with multiple processors can be classified as *tightly-* or *loosely-coupled*. In tightly-coupled systems, memory is accessible to all processors and communication is performed through shared data, while in loosely-coupled systems processes run in disjoint address spaces and communicate by sending explicit messages to one another. On the software side, many efforts have been undertaken to create a standard parallel processing environment, such as parallelizing compilers, libraries, language extensions and tools to manage parallel resources and to identify program errors and performance bottlenecks. PVM and MPI have emerged to be widely used in the realm of message passing libraries mostly because they address portability, heterogeneity, and scalability issues and both tend to be a standard for programming parallel distributed memory machines.

The main difference between them is that MPI is a specification which hardware vendors can implement on their parallel machines in order to achieve high throughput and low latency in message exchange. MPI has a large variety of point-to-point and group communication primitives and allows the programmer to specify a logical communication topology. On the other hand, PVM is designed to run in heterogeneous environments and provides dynamic process control and some resource management facilities. The central notion in PVM is a *virtual machine*, a pool of heterogeneous machines connected by

a network that can dynamically be reconfigured both in terms of machines and parallel tasks. These characteristics, coupled with the possibility of creating special tasks which can intercept PVM library calls, are the reason why our first implementation was PVM based, especially since the MPI specification (MPI-1) at the time of our development did not support any dynamic host or process facilities.

2.1. Parallel Virtual Machine

The PVM system presents a general library-based message passing interface to enable distributed memory computing on parallel computers as well as on heterogeneous workstations and PC clusters. With thousands of users PVM has become the de facto standard and widely prevalent parallel programming paradigm. Starting tasks in parallel, message exchange among tasks as well as synchronization, virtual machine management, process management and other miscellaneous functions are accomplished by the PVM library in conjunction with PVM daemons that run on each computing node and cooperate to emulate a parallel machine. With reference to Figure 1, where the structure of PVM is depicted by making use of the UML [2] class diagram, the first PVM daemon, called *master*, is started manually and the others (*slaves*) are started by the master on a user-defined host pool using remote login, remote shell program or the `rexec` system call.

Parallelizing an application based on functional and data parallelism results in several cooperating PVM *tasks* that can run in parallel, synchronize and send messages to one another. Tasks can be dynamically started and destroyed by calling the `pvm_spawn` and `pvm_kill` library functions, respectively. The PVM library then forwards the requests to the daemon specified in the parameter list. If a special task called *tasker* is not registered, new tasks are started and killed by the daemon calling the appropriate UNIX system calls. If the tasker runs, the daemon forwards the requests to the tasker which controls the execution of the tasks running on its host. The tasker is the direct parent of any process it controls and can be used to implement a distributed debugging system.

Interprocess communication in PVM is accomplished with the `pvm_send` and `pvm_recv` calls that support messages containing multiple data types, but require explicit encoding and decoding calls for buffer construction and extraction. Without setting the options that allow message exchange to use direct process to process stream

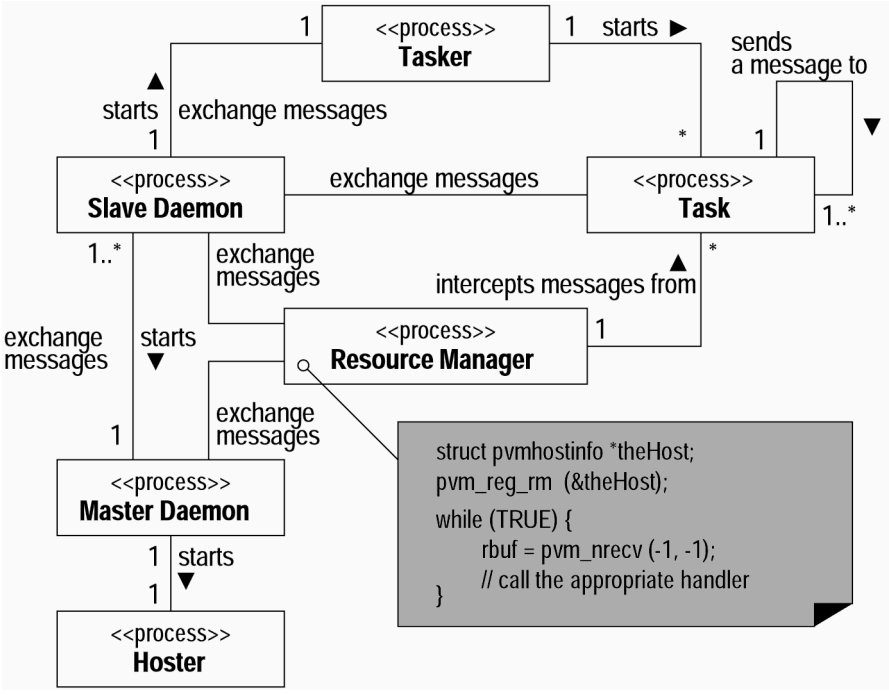

Fig. 1. The Structure of the PVM System.

connection, i.e. TCP, data is transfered via daemons which operate on top of the UDP protocol. Messages are reliably delivered and buffered, so a task can use both blocking and nonblocking calls to receive messages. Each message is labeled with a user-supplied message tag and with a system-supplied context tag. The message tag and the sender identifier are used to discriminate among multiple messages arriving at the same time. The context tag discriminates among libraries which use PVM message passing capabilities and are linked together in the same executable. PVM provides dynamic and static group operations, which are useful when more than one sender or receiver is involved in message exchange or collaborative computation.

Any task can dynamically configure the virtual machine by calling `pvm_addhosts` and `pvm_delete` calls to the library which forwards those requests to the master daemon. In order to prevent system inconsistency the master daemon uses a two phase commit protocol to set up the new virtual machine, but since any task can reconfigure the virtual machine at any time, races are possible. As the tasker helps a daemon to manage the tasks, another special task, named hoster, can be registered by the master daemon to control the hosts' configuration.

By default, the task and host scheduling in PVM is accomplished by a simple round robin algorithm. Beginning with version 3.3, PVM supports a resource manager interface, which allows special tasks to be registered by the system and be responsible for the task and host placement decisions. If a *resource manager* is registered, each library call that manipulates processes or hosts is *intercepted* by the PVM library and sent to the resource manager task using special messages. Typically, one resource manager is registered for an entire virtual machine, but each daemon can have its own. The daemons without resource manager are handled by the resource manager associated with the master daemon. Any task can be registered with PVM as a tasker, a hoster or a resource manager by calling `pvm_reg_tasker`, `pvm_reg_hoster` or `pvm_reg_rm` library call, respectively.

3. Resource Management and Load Balancing

Resource management and load balancing are very important techniques to efficiently utilize the resources in a NOW and to minimize the execution time of parallel applications. Taking scheduling decisions and controlling resources and applications in such a way that the load is evenly distributed are two major issues that must be taken into account with the design and implementation of resource management systems. The mapping between hosts and application tasks is performed by a *scheduling algorithm*, which typically selects an application waiting in the batch queue, configures a runtime environment for the application, and starts it up.

The scheduling algorithms require static information about hardware configuration and host's performance data as well as dynamic information about the actual resource utilization and load distribution in the NOW. Besides the information about the hosts, the scheduler also needs information about the applications' resource requirements. Usually, resource requirements must be specified by the user and given in terms of such issues as the size of main memory, machine architecture, minimum and maximum number of nodes, and the host running the master process. There are a number of scheduling algorithms covering various management and mapping strategies, but a thorough treatment of these issues is out of the scope of this work.

A *scheduler* in a resource management system is not only responsible for the initial placement of applications but also for load balancing which requires a remapping of applications at runtime. Load balancing is crucial for message passing applications because the slowest process could determine the execution time of the whole parallel application and, thus, defer the synchronization between the processes, thereby yielding poor performance. Therefore, cooperating processes in a parallel application should be started on hosts with comparable performance and load so that synchronization delays can be minimized and the load situation remains in a balanced state.

The basis for load balancing algorithms is information about the load distribution in the NOW that is often characterized by load values, e.g.

the average length of the CPU run queue or the CPU utilization of the running processes. The load is measured periodically and load evaluation algorithms create a load map of the system by comparing the load values with each other or with predefined threshold values. Load balancing is a complex optimization problem with a given cost function to equalize the load on the hosts. This optimization problem is NP-complete and has to be solved by using suboptimal or heuristic algorithms. A survey of load balancing algorithms is presented in [19].

An unbalanced load distribution in a NOW requires load migration by, for instance, remapping processes to other machines. Important issues in process migration are the migration costs and the costs for the reconfiguration of the virtual machine, since both are quite expensive and produce additional load. Process migration should be avoided, if the process finishes soon, but predicting the remaining runtime of a process is a difficult problem [9]. A further reason for process migration is the interactive use of target hosts. As a NOW usually is non-dedicated, interactive users expect to have priority over resource-intense applications that are primarily intended to use idle resources. If an interactive user starts working on a host executing parallel processes, the processes must be migrated to an idle host to ensure the interactive user a reasonable response time.

The most prevalent facility for the migration and the checkpointing of a single process is the Condor [12] single checkpointer that is available for almost every hardware platform. Condor's job is able to transparently vacate a workstation when the user attempts to use it. In this case, Condor will either transparently migrate the job to another idle workstation or just keep it in a queue until an idle workstation has been found. Clearly, the computation already performed should not be sacrificed and the job has to continue execution from the point where it was forced to vacate the machine. Some hardware vendors provide their own checkpointing mechanisms for sequential processes, e.g. the user-transparent Checkpoint-Restart mechanism of SGI [23]. Process migration in message passing parallel applications is more complex than in the sequential ones, mostly because parallel processes communicate with each other and process migration has to deal with the communication routes. At the LRR-TUM the CoCheck [20, 21] system **Co**nsistent **Check**points) has been developed for process migration in message passing environments while Coral [25] migrate processes in software DSM (Distributed Shared Memory) environments.

Scheduling algorithms in existing resource management systems are only able to perform the initial task-host placement for parallel applications. Dynamic load balancing at runtime is not supported because of the missing control over parallel processes. Sequential applications are started and are fully controlled by the resource management system. Within parallel applications, only the first process is started by the resource management system, while other processes are started by the application's tasks or by the message passing runtime system and therefore are unknown to the resource management system.

Full control over parallel processes is a crucial issue for resource management systems because without full control, they are not able to kill the whole application or to migrate processes. Furthermore, full control over parallel applications is required for writing periodic checkpoints in order to be able to continue the execution of an application after a failure, without restarting it form the very beginning. Finally, resource limitations can only be set and controlled if the resource management system knows of each process.

4. The SEMPA Resource Manager

SEMPA (**S**oftware **E**ngineering **M**ethods for **P**arallel **A**pplications in Scientific Computing) is a research project funded by the German Federal Department of Education, Science, Research, and Technology to define software engineering methods for the parallelization of scientific computing applications [11, 14, 13]. One of the aims in the SEMPA project is to design and implement a resource management system with full support for message passing applications. The SEMPA Resource Manager has been developed to connect the PVM message passing environment and the CODINE resource management system.

CODINE is a batch queuing and resource management system that supports the execution of sequential and parallel applications in heterogeneous NOWs. It is developed and distributed by GENIAS [6]. All the information about hosts and their utilization and jobs and their status is managed by the *qmaster* process, which is CODINE's central component. Using this information and the user defined resource requirements, the CODINE scheduler *qschedd* determines the host pool where the job is to be computed. The CODINE *execd* running on every host in the

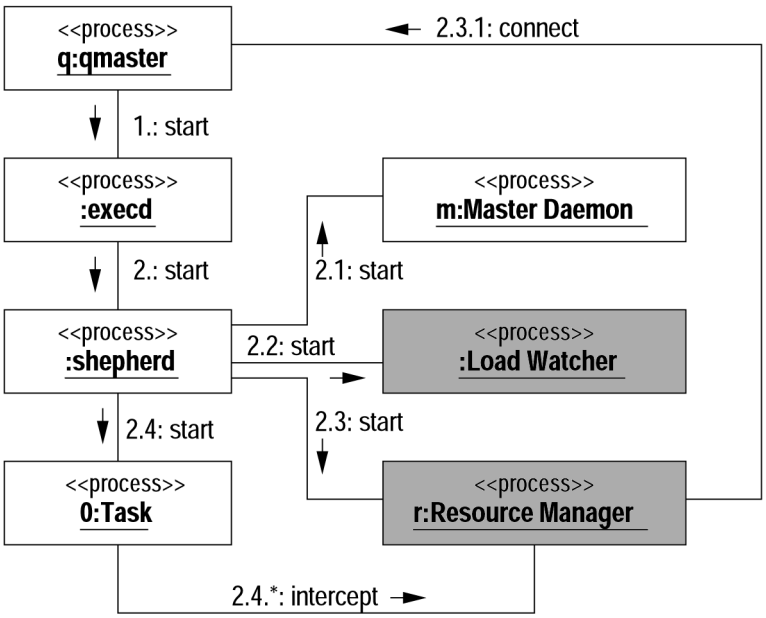

Fig. 2. The Structure of SEMPA Resource Manager on the Master Host.

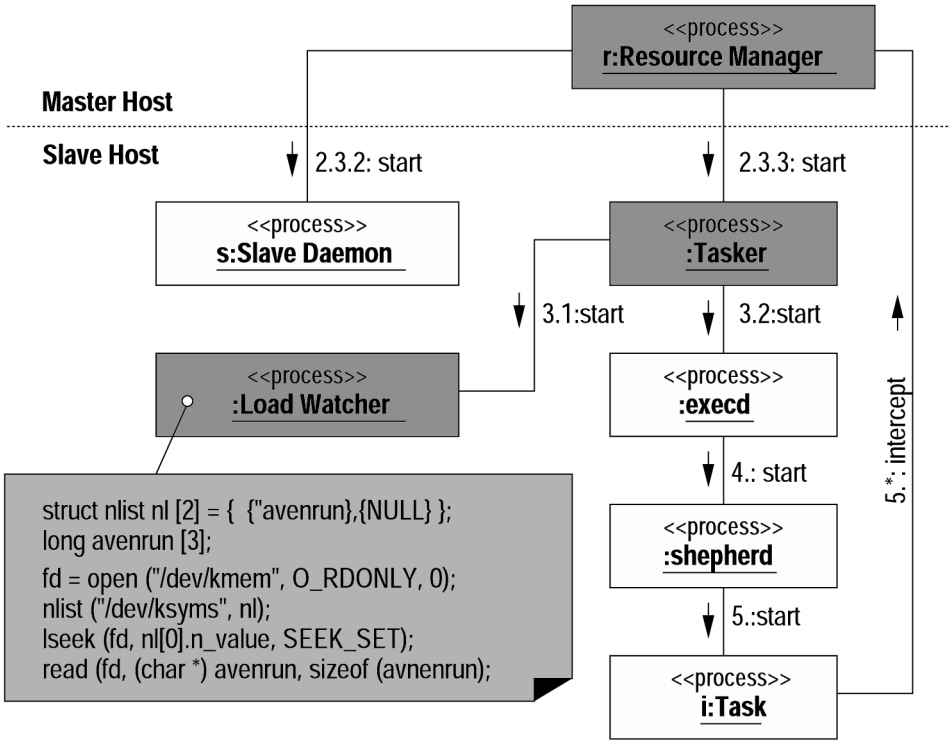

Fig. 3. Structure of SEMPA Resource Manager on a Slave Host.

NOW measures the load on the host periodically and controls the execution of jobs on the host in cooperation with *shepherd* processes.

With reference to Figures 2 and 3, where the structure of our resource management approach for both the master host and slave hosts is depicted, we make use of the PVM resource manager interface to connect CODINE and PVM. We also integrate some additional functionality into our resource management scheme, e.g. load management including load evaluation and load migration capabilities [15, 16].

The basis for load evaluation algorithms are load values that can be received from the CODINE *execd* or an external load measurement component. However, we have implemented a load watching component, called *load watcher*, in order to faster react on the load changes in the virtual machine because obtaining the load values measured by the CODINE *execd* via the CODINE *qmaster* takes more time and since we did not want to directly interact with *execd* .

The SEMPA Resource Manager makes use of CoCheck for process migration. CoCheck is a protocol that writes consistent yet transparent checkpoints of a parallel application based on the message passing paradigm and uses the checkpointing facilities of Condor to write the state of a single process to a file or a socket. During process migration, the checkpoint state is transferred to the target host and restarted with the same state as it was before the migration. Since the process selected to be migrated communicates with other processes, all affected processes must be stopped and all corresponding communication routes must be destroyed. However, after migration, these routes must be rebuilt and the processes affected by migration must be able to continue execution.

In Figures 2 and 3 we make use of UML object diagrams to depict the overall structure of the SEMPA Resource Manager as well as to show the interaction patterns between its components. The chronological sequence of actions, performed to start the system up, is indicated by sequence numbers from 1 to 5. Before a PVM application (a group of *tasks*) is started, its runtime environment, according to the user requirements, must be configured. When the application is selected by the CODINE scheduler, the runtime configuration is passed to the PVM *resource manager* that builds up the runtime environment. The first process $(0 : Task)$ of the parallel application is then started by the CODINE *execd* and controlled by a *shepherd* process. Other application processes are spawned when a pvm task calls the `pvm_spawn` routine. This routine, in our scheme, is intercepted by the PVM resource manager which then selects the host on which to start the requested task.

The SEMPA Resource Manager has three options for host selection. It can utilize the PVM default option, e.g. a simple round robin mechanism without taking into account the current load distribution. Furthermore, the PVM resource manager may consult CODINE to select the host considering the load distribution in the whole system controlled by CODINE. Finally, our resource manager is extended with a load evaluation component, which may select an appropriate host within the actual runtime environment. If there is no appropriate host in the actual runtime environment, the PVM resource manager may requests a new host from CODINE. After a host has been selected, the PVM *tasker* on the selected host creates the task $(i : Task)$ in conjunction with the CODINE *execd* and a new *shepherd* process.

When the process terminates, the *shepherd* process passes information to the PVM tasker which forwards it to the PVM resource manager. Resource statistics about the process are collected by the *shepherd* process and handed on to the *execd* which periodically delivers it to the CODINE *qmaster* . PVM daemons are started with an application and terminate when the application finishes.

5. Measurements and Evaluation

The primary objective of the measurement studies was to compare the overheads in the startup of the PVM virtual machine and in the library calls caused by the added resource management functionalities. The testbed consisted of heterogeneous workstations (SUN and SGI) connected by Ethernet. The first group of measurements dealt with starting of the PVM virtual machine and its dynamic reconfiguration with and without the PVM resource manager and performance results are depicted in Figure 1.

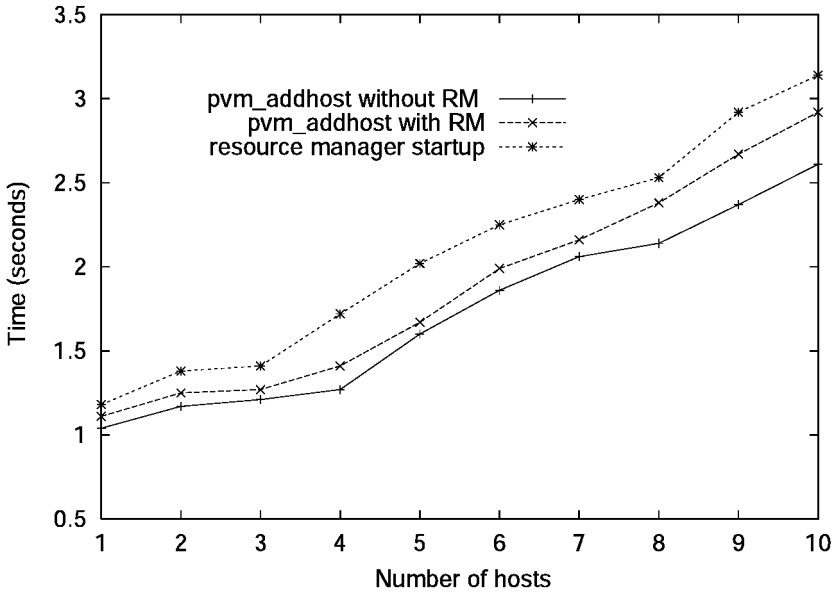

Fig. 4. Adding Hosts to the PVM Environment – Comparison.

The curve marked *"pvm_addhost without RM"* indicates the time needed to enhance the PVM virtual machine calling *pvm_addhost* library call from an application task, but, without the resource management layer. The same test is repeated, but this time with the resource manager layer between tasks and the PVM runtime system. The graphs show that the performance is only slightly diminished by the improved functionality and that the average overhead is about 10-30 %. For the curve marked *"resource management startup"*, the measurement is performed in the resource manager. However, this time the virtual machine is built by the resource manager at the startup and the time getting the virtual machine started, including RM initialization time is about 20 % bigger than the one when the virtual machine is dynamically enhanced.

Table 1 shows the duration of the PVM `pvm_spawn` call in different scenarios; on the local or remote machine with and without the PVM resource management layer. In this test, especially for the remote spawn the overhead is more than 100%. This is primarily caused by the algorithm for the task-host matching used in the resource manager. It would be even more pronounced if the resource manager cooperated with the CODINE to get the best host in the whole system. Presently, our resource manager selects the best host from the host set received from CODINE at the startup, but this overhead in negligible in comparison with the benefit from starting a long-running task is on the appropriate machine.

The load situation in a NOW must be permanently controlled in order to have an insight into the resource utilization and to detect load imbalance. Our load measurement component runs at each host in the virtual machine and measures the load at regular time intervals (i.e. 5 seconds). The load on SGI machines is measured in terms of the CPU utilization and takes about 170 μs. On SUN machines the load is measured as the average length of the CPU run queue (*avenrun*) and takes about 10 μs. Load measurement on SGI machines is obtained using the `/proc` file system, while the one on SUN machines is based on the `/dev/kmem` file that contains an image of the kernel virtual mem-

Spawn Time (*ms*)	local host	remote host
without the SEMPA Resource Manager	24	35
with the SEMPA Resource Manager	42	93

Table 1. Starting of a PVM process.

Time (μs)	Measurement	Evaluation	Sending
SUN host	10	0.38	0.78
SGI host	170	0.21	0.61

Table 2. Management of Load Values.

ory of the computing node. As an example of load measurement, we show in Figure 2 a code snippet which obtains the average length of the running queue from the Solaris kernel. The address of these three long integers can be, using the `nlist` function, found in the `/dev/ksyms` file which contains kernel symbols and their actual addresses.

If the load situation changes or if an interactive user starts working, a PVM message is sent to the PVM resource manager, which migrates processes using CoCheck mechanisms based on Condor checkpoint files. The time needed to migrate a process depends on the process size, network bandwidth and latency and on the number of communication routes the process has with another PVM tasks. The performance results for process migration using CoCheck can be found in [21]. Table 1 shows the time needed for the load measurement and evaluation as well as the time for sending the load value to the PVM resource manager. The time needed for the load evaluation and sending can is negligible in comparison with the time needed for its measurement.

6. Discussion and Future Work

While the design and implementation of the SEMPA Resource Manager have proven to be successful in controlling and rescheduling parallel processes, we intend to continue to enhance our project along several lines.

One of the first enhancements to be undertaken is the cooperation between the PVM resource manager and CODINE in allocating resources for a parallel application. Presently, the PVM resource manager only gets a host pool from CODINE at startup but the virtual machine cannot be extended automatically at runtime. The second issue is heterogeneous process migration. Currently, CoCheck is only able to migrate processes to machines with the same processor and the same operating system version. Finally, our implementation is PVM based mostly because at the time of our implementation, the MPI standard (MPI-1) did not support any routine for dynamic host or process manipulation which is crucial for our implementation scheme. The only way was to extend an existing MPI implementation by non-standard functions to support resource management what would circumvent the standardization efforts of the MPI Forum. The improved standard MPI-2 offers some interfaces for user-defined extensions and primitives for dynamic process management [4]. A further interesting function makes it possible to build up a connection between processes in different MPI applications. This function could be used to implement a daemon with the same functionality as the PVM resource manager.

The SEMPA Resource Manager has many advantages over other resource management systems which in many cases do not have control over parallel applications. For instance, the PSCHED standard proposal aims at standardizing the interaction between the different components involved in the scheduling of parallel applications [5]. Interfaces are defined among a resource manager, a message passing system and a scheduler to allow the cooperation of different systems and to make it easier to exchange components with standardized interfaces. PSCHED is just an interface standard, and, at the time of writing, an implementation is not available as yet.

CARMI (**C**ondor **R**esource **M**anagement **I**nterface) is an interface between Condor and PVM to bring together the functionalities of a resource management system and a message passing environment [18]. The PVM functions are divided into a group of communication calls and a group of resource management calls. With the use of the PVM resource manager, the resource management functions of PVM are executed by Condor. In contrast to the SEMPA Resource Manager CARMI only supports resource allocation but not scheduling or dynamic load balancing at runtime.

The Prospero Resource Manager supports resource allocation and scheduling in large networks and multiprocessor systems [17]. The basic concept of the Prospero Resource Manager is similar to the SEMPA Resource Manager : a job manager controls the resources of a parallel application and requests resources from a system manager on demand. A major drawback of the Prospero Resource Manager is that a process migration component is missing.

Approaches similar to the SEMPA Resource Manager are utilized by the Coral [26, 24] project. Coral (**C**ooperative **O**nine Monito**r**ing **A**ctions **L**ayer) is aimed at the design and implementation of online monitoring systems for applications based on the DSM programming paradigm. Coral only instruments parallel activities and constructs while the sequential ones are intended to be included by utilizing legacy sequential software. Coral is primarily focused on the interaction among parallel applications and DSM runtime systems, the transparent management of DSM mechanism including process migration and consistent checkpointing, and the consistency of monitoring actions in a multiple-tool environment, but, in contrast to the SEMPA Resource Manager , does not deal with scheduling strategies.

7. Conclusion

Message passing environments support the parallelization of applications and provide runtime environments for their execution. Resource management functionalities are especially required in NOWs to configure the runtime environment and schedule the applications to appropriate hosts. Load balancing is an extremely important issue for parallel applications to avoid waiting times potentially caused by synchronizations among parallel activities. However, there is no interface to make the information of a resource management system available to a message passing environment.

In this article, we discussed the problems arising from the missing cooperation between resource management systems and message passing environments and presented the SEMPA Resource Manager to overcome these deficiencies. The performance measurements show that there is negligible overhead caused by the cooperation. Up to now, there is only an implementation of the SEMPA Resource Manager for PVM because the current (at the time of our development) MPI-1 standard lacks dynamic host and process management facilities and misses external interfaces. The MPI-2 standard provides the functionalities required for an interface to a resource management system, however, only part of MPI-2 has already been implemented and is available as a message-passing environment.

References

[1] A. BEGUELIN AND J. DONGARA AND A. GEIST AND R. MANCHEK AND W. JIANG AND V. S. SUNDERAM, *PVM: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[2] G. BOOCH AND J. RUMBAUGH AND I. JACOBSON, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[3] *The MPI Forum. MPI: A Message-Passing Interface Standard, Version 1.1*, Tehnical Report, University of Tennessee, Knoxville, TN, June 1995. `http://www.mpi-forum.org/docs/mpi-11.ps.Z`

[4] *The MPI Forum. MPI-2: Extensions to the Message-Passing Interface*, Tehnical Report, University of Tennessee, Knoxville, TN, July 1997.

[5] D. G. FEITELSON AND L. RUDOLPH AND U. SCHWIEGELSHOHN AND K. C. SEVCIK AND P. WONG, Theory and Practice in Parallel Job Scheduling, In *IPPS'97 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291, *of Lecture Notes in Computer Science*, pages 1–34, Springer-Verlag, 1997. `http://www.cs.huji.ac.il/~feit/parsched.html`

[6] GENIAS Software GmbH, D-93073 Neutraubling, Germany. *CODINE Manual, Version 4.0.2*, 1997.

[7] W. Gropp and E. Lusk and N. Doss and A. Skjellum, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Tehnical Report, Argonne National Lab, July 1996.

[8] T. P. GREEN AND J. SNYDER, *DQS, A Distributed Queuing System*, Technical Report FSU-SCRI-92-115, Supercomputer Computations Research Institute, Florida State University, 1992.

[9] M. HARCHOL-BALTER AND A. B. DOWNEY, Exploiting Process Lifetime Distributions for Dynamic Load Balancing, in *1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, USA, pages 13–24. ACM Press, 1996.

[10] J. P. JONES, *NAS Requirements Checklist for Job Queuing/Scheduling Software*, Technical Report NAS-96-003, NAS High Performance Processing Group, NASA Ames Research Center, Moffett Field, CA, April 1996. http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-96-003.

[11] P. LUKSCH AND U. MAIER AND S. RATHMAYER AND M. WEIDMANN AND F. UNGER, Software Engineering Methods for Parallel Applications in Scientific Computing – Project SEMPA. *IEEE Concurrency*, pages 64–72, July-September 1997. http://wwwbode.informatik.tu-muenchen.de/archiv/artikel/ieee-concurrency/sempa.ps.gz

[12] M. LITZKOW AND T. TANNENBAUM AND J. BASNEY AND M. LIVNY, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Environment*, Technical Report 1346, University of Wisconsin-Madison, Computer Sciences Department, 1997. http://www.cs.wisc.edu/condor/publications.html

[13] U. MAIER, *Konzepte zur Ressourcenverwaltung für wissenschaftlich-technische Anwendungen in verteilten Rechensystemen*, PhD Thesis, Technische Universität München, 1999.

[14] U. MAIER AND P. LUKSCH AND A. BODE, Experiences with the SEMPA Resource Manager for Real World Scientific Computing in Networks of Workstations, *Parallel and Distributed Computing Practices*, 1999. to appear.

[15] U. MAIER AND G. STELLNER AND I. ZORAJA, Batch Queuing and Resource Management for PVM Applications in a Network of Workstations, in *Mitteilungen – Gesellschaft f ur Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen*, volume 16, pages 11–20, Gesellschaft f ur Informatik e.V., Parallel-Algorithmen, Rechenstrukturen und Systemsoftware, December, 1997. http://wwwbode.informatik.tu-muenchen.de/archiv/artikel/pars97/maier.ps.gz

[16] U. MAIER AND G. STELLNER AND I. ZORAJA, Resource Allocation, Scheduling and Load Balancing based on the PVM Resource Manager, in *Parallel Computing: Fundamentals, Applications and New Directions*, volume 12, pages 711–718, Elsevier Publishers, 1998. http://wwwbode.informatik.tu-muenchen.de/archiv/artikel/parco97/maier.ps.gz

[17] B. C. NEUMAN AND S. RAO, The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems, *Concurrency: Practice and Experience*, 6(4):339–355, June 1994. ftp://prospero.isi.edu/pub/papers/prm

[18] J. PRUYNE AND M. LIVNY, Interfacing Condor and PVM to harness the cycles of workstation clusters, *Future Generations of Computer Systems*, 12(1):67–85, May 1996. http://www.cs.wisc.edu/condor/publications.html

[19] B. A. SHIRAZI AND A. R. HURSON AND K. M. KAVI, Scheduling and Load Balancing in Parallel and Distributed Systems, *IEEE Computer Society Press*, 1995.

[20] G. STELLNER AND J. PRUYNE, Resource Management and Checkpointing for PVM, In *Proceedings of EuroPVM'95*, volume 5, pages 131–136, Hermès, September 1995.

[21] G. STELLNER. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, IEEE Computer Society Press, April 1996.

[22] V. S. SUNDERAM, PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[23] BILL TUTHILL, *IRIX Checkpoint and Restart™ Operation Guide*, Silicon Graphics, Inc., 1996. http://techpubs.sgi.com/library/lib/makepage.cgi?007-3236-001

[24] I. ZORAJA AND A. BODE AND V. SUNDERAM, A Framework for Process Migration in Software DSM Environments, Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, pages 158–165, *IEEE Computer Society*, January 2000.

[25] I. ZORAJA, *Online Monitoring in Software DSM Systems,* PhD Thesis, Technische Universit at M unchen, submitted, March 2000.

[26] I. ZORAJA AND G. RACKL AND T. LUDWIG. Towards Monitoring in Parallel and Distributed Systems, in *Proceedings of SoftCOM '99*, pages 133–141. FESB Split, October 1999.

Received: March, 1998
Revised: May, 2000
Accepted: February, 2001

Contact address:
Ivan Zoraja
FESB,
Department of Electronics and Computer Science
University of Split
21000 Split
Croatia
e-mail: {zoraja|pslap}@fesb.hr

Ursula Seitz
LRR-TUM
Institut f ur Informatik
Technische Universit at M unchen
80290 M unchen
Germany

Arndt Bode
LRR-TUM
Institut f ur Informatik
Technische Universit at M unchen
80290 M unchen
Germany
e-mail: {maier|bode}@in.tum.de

Petar Slapničar
FESB,
Department of Electronics and Computer Science
University of Split
21000 Split
Croatia

DR. IVAN ZORAJA obtained his diploma degree in electronics from FESB in Split, his master degree in computer science from FER in Zagreb, both in Croatia, and his Ph.D degree (Dr. rer. nat) in computer science from the Technical University of Munich in Germany. His education includes several research stays in Germany (three years) and USA (two years). He is currently employed at FESB in Split in Croatia. His primary research interests are parallel and distributed systems including distributed shared memory systems and distributed object computing.

DR. URSULA SEITZ obtained her diploma degree (Dipl. Inform) in 1994 and her Ph.D (Dr. rer. nat) in 1999 both from the Technical University of Munich. The title of her PhD thesis is "Resource Management for Scientific Computing". She is currently at Kratzer Automation AG, working on software development for pthe automation of industrial processes. Her major points of interest are parallel applications, distributed systems, and metacomputing.

PROF. DR. ARNDT BODE obtained his diploma degree (Dipl. Inform.) in 1972 and his Ph.D (Dr. rer.nat.) in 1975 both from the Technical University of Karlsruhe, his Dr.-Ing. habil in 1985 from the University of Erlangen-N urnberg in Germany. He used to work for the Universities of Gie sen and Erlangen and is a full professor in computer science at the Technical University of Munich since 1987 and also a vice-president since 1999. His main research interests are computer architecture, microprocessing, and parallel and distributed systems. He is the author and co-author of more than 175 publications including 10 books about the said topics.

PROF. DR. PETAR SLAPNIČAR obtained his diploma degree (Dipl. Ing.) in 1957, his master degree in 1964, and his Ph.D in 1977 all from the Faculty of Electrical Engineering in Zagreb, Croatia. Since 1978 he is a full professor at the Faculty of Electrical Engineering, Mechanical Engineering and, Naval Architecture in Split, Croatia. His main research interests are pulse and digital electronics, CAD for electronic devices, and parallel and distributed computing. He is the author and co-author of more than 50 scientific publications.